Homework 6 ME333 - Winter 2025

Zhengyang Kris Weng Submission 02/10/2025

Chapter 5.

3. Build the program with no optimization and look at the disassembly. Answer the following questions.

a. Which combinations of data types and arithmetic functions result in a jump to a subroutine? From your disassembly file, copy the C statement and the assembly commands it expands to (including the jump) for one example.

```
    j3 = j1/j2;
9d0086bc:   8fc60028    lw  a2,40(s8)
9d0086c0:   8fc7002c    lw  a3,44(s8)
9d0086c4:   8fc40020    lw  a0,32(s8)
9d0086c8:   8fc50024    lw  a1,36(s8)
9d0086cc:   0f401e52    jal 9d007948 <__divdi3>   <----------- Jumps here
9d0086d0:   00000000    nop
9d0086d4:   afc20050    sw  v0,80(s8)
9d0086d8:   afc30054    sw  v1,84(s8)

    9d007948 <__divdi3>:   <----------------------------------- Jumps to
9d007948:   04a10007    bgez    a1,9d007968 <__divdi3+0x20>
9d00794c:   00005025    move    t2,zero
9d007950:   00041023    negu    v0,a0
9d007954:   0002182b    sltu    v1,zero,v0

... omitting 200 lines here...

9d007d78:   1080ff3e    beqz    a0,9d007a74 <__divdi3+0x12c>
9d007d7c:   00000000    nop
9d007d80:   2442ffff    addiu   v0,v0,-1
9d007d84:   1000ff3b    b   9d007a74 <__divdi3+0x12c>
9d007d88:   00001825    move    v1,zero
```

b. For those statements that do not result in a jump to a subroutine, which combination(s) of data types and arithmetic functions result in the fewest assembly commands? From your disassembly, copy the C statement and its assembly commands for one of these examples. Is the smallest data type, char , involved in it? If not, what is the purpose of extra assembly command(s) for the char data type vs. the int data type?

```
    i3 = i1+i2;
9d0085d0:   8fc30014    lw  v1,20(s8)
9d0085d4:   8fc20018    lw  v0,24(s8)
```

```
9d0085d8:   00621021    addu    v0,v1,v0
9d0085dc:   afc2004c    sw  v0,76(s8)
```

The `int` data type takes the fewest assembly commands. Extra steps in the `char` data type is doing memory operations since the `char` is shorter, and it takes a step to mask to only keep the lower 8 bits for the operation.

c. Fill in the following table. Each cell should have two numbers: the number of assembly commands for the specified operation and data type, and the ratio of this number (greater than or equal to 1.0) to the smallest number of assembly commands in the table.

```
          char          int          long long        float      long double
    +   1.25 (5)     1.0 (4)     2.75 (11)           J              J
    -   1.25 (5)     1.0 (4)     2.75 (11)           J              J
    *   1.25 (5)     1.0 (4)     4.50 (18)           J              J
    /   1.75 (7)     1.75 (7)        J               J              J
```

d. From the disassembly, find out the name of any math subroutine that has been added to your assembly code, and find their mappings in .map file.

This part of the disassembly file contains, and they map to:

```
__addsf3 --> 0x000000009d008a9c
__subsf3 --> 0x000000009d008a94
__mulsf3 --> 0x000000009d008f3c
__divsf3 --> 0x000000009d008d0c

__divdi3 --> 0x000000009d007948

__adddf3 --> 0x000000009d007d94
__subdf3 --> 0x000000009d007d8c
__muldf3 --> 0x000000009d0081bc
__divdf3 --> 0x000000009d007490
```

4. Let us look at the assembly code for bit manipulation. How many commands does each use? For unsigned integers, bit-shifting left and right make for computationally efficient multiplies and divides, respectively, by powers of 2.

```
    u3 = u1 & u2; // bitwise AND    <------ 4 commands
9d0078d0:   8fc30000    lw  v1,0(s8)
9d0078d4:   8fc20004    lw  v0,4(s8)
9d0078d8:   00621024    and v0,v1,v0
9d0078dc:   afc20008    sw  v0,8(s8)
    u3 = u1 | u2; // bitwise OR     <------ 4 commands
9d0078e0:   8fc30000    lw  v1,0(s8)
```

```
9d0078e4:   8fc20004    lw  v0,4(s8)
9d0078e8:   00621025    or  v0,v1,v0
9d0078ec:   afc20008    sw  v0,8(s8)
    u3 = u2 << 4; // shift left 4 spaces, or multiply by 2^4 = 16   <------
3 commands
9d0078f0:   8fc20004    lw  v0,4(s8)
9d0078f4:   00021100    sll v0,v0,0x4
9d0078f8:   afc20008    sw  v0,8(s8)
    u3 = u1 >> 3; // shift right 3 spaces, or divide by 2^3 = 8     <------
3 commands
9d0078fc:   8fc20000    lw  v0,0(s8)
9d007900:   000210c2    srl v0,v0,0x3
9d007904:   afc20008    sw  v0,8(s8)
```

however, on the side note... I also did multiplication and division normally but they seem to take the same amount of commands? u3 = u2 * 16; <------ 3 commands 9d00788c: 8fc20004 lw v0,4(s8) 9d007890: 00021100 sll v0,v0,0x4 9d007894: afc20008 sw v0,8(s8) u3 = u2 / 8; <------ 3 commands 9d007898: 8fc20004 lw v0,4(s8) 9d00789c: 000210c2 srl v0,v0,0x3 9d0078a0: afc20008 sw v0,8(s8)

Not sure why this is the case ¯_(ツ)_/¯. I have optimization set at -O0.

Chapter 6

1. Interrupts can be used to implement a fixed frequency control loop (e.g., 1 kHz). Another method for executing code at a fixed frequency is polling: you can keep checking the core timer, and when some number of ticks has passed, execute the control routine. Polling can also be used to check for changes on input pins and other events. Give pros and cons (if any) of using interrupts vs. polling.

Interrupts

Pros:

- Efficient Resource Utilization: Interrupts allow the microcontroller to perform other tasks when not handling an event, conserving CPU cycles.
- Real-Time Response: Events are handled immediately upon occurrence, ensuring timely responses without delays.
- Multitasking Capability: The system can execute multiple tasks efficiently by interrupting only when necessary.

Cons:

- Complexity and Overhead: Interrupts require setup of ISRs and can complicate program flow, especially with multiple or simultaneous events.
- Latency Issues: Interrupt handling can introduce delays (latency) due to context switching, affecting precise timing.
- Potential Instability: Frequent or prolonged interrupts might destabilize the system if not managed well.

Polling

Pros:

- Simplicity: Easier to implement with straightforward loops and no need for ISR setup.
- Predictable Timing: Each check takes a known amount of time, which can be beneficial for critical timing applications.
- Low Overhead: Avoids context switching and setup overhead associated with interrupts.

Cons:

- High CPU Usage: Continuous checking wastes CPU cycles when events are infrequent.
- Inefficient Power Use: Constantly running loops drain power in battery-powered devices.
- Response Delay: Events occurring between polls may experience delayed response times.

4.

a. What happens if an IRQ is generated for an ISR at priority level 4, subpriority level 2 while the CPU is in normal execution (not executing an ISR)?

The CPU immediately services the priority 4, subpriority 2 interrupt.

b. hat happens if that IRQ is generated while the CPU is executing a priority level 2, subpriority level 3 ISR?

The CPU preempts the level 2 ISR and services the level 4 ISR; afterward, it returns to the level 2 ISR.

c. What happens if that IRQ is generated while the CPU is executing a priority level 4, subpriority level 0 ISR?

The new interrupt (level 4, subpriority 2) is not serviced immediately. It waits until the CPU completes the current level 4 ISR.

d. What happens if that IRQ is generated while the CPU is executing a priority level 6, subpriority level 0 ISR?

The level 4 interrupt cannot preempt the level 6 ISR. It is deferred until the level 6 ISR finishes.

5. An interrupt asks the CPU to stop what it’s doing, attend to something else, and then return to what it was doing. When the CPU is asked to stop what it’s doing, it needs to remember “context” of what it was working on, i.e., the values currently stored in the CPU registers.

a. Assuming no shadow register set, what is the first thing the CPU must do before executing the ISR and the last thing it must do upon completing the ISR?

First thing (on ISR entry) The compiler-generated ISR “prologue” must save any working registers that the ISR plans to use (including the return address ra, frame pointer s8, any callee-saved registers, etc.) onto the stack so that the interrupt cannot clobber the state being used by the interrupted code.

Last thing (on ISR exit) The ISR “epilogue” must restore all those saved registers (pop them back from the stack) and then execute the special “return from exception” instruction (on MIPS, eret) to resume whatever the CPU was doing before the interrupt.

b. How does using the shadow register set change the situation?

With a shadow register set, when the CPU takes an interrupt at a certain priority that qualifies for shadow register usage, it automatically switches to a different physical set of registers. This means the ISR can use those registers freely without corrupting the registers that the main program was using.

As soon as the interrupt exits, the hardware switches back to the original register set, so there is no need for the software overhead of saving and restoring registers on the stack. This can make ISRs much faster and more efficient.

8. For the problems below, use only the SFRs IECx, IFSx, IPCy, and INTCON, and their CLR, SET, and INV registers (do not use other registers, nor the bit fields as in IFS0bits.INT0IF). Give valid C bit manipulation commands to perform the operations without changing any uninvolved bits. Also indicate, in English, what you are trying to do, in case you have the right idea but wrong C statements. Do not use any constants defined in Microchip XC32 files; just use numbers.

a. Enable the Timer2 interrupt, set its flag status to 0, and set its vector's priority and subpriority to 5 and 2, respectively.

```
// Clear the T2IF bit (bit 9) in IFS0
IFS0CLR = 1 << 9;

// Enable the T2IE bit (bit 9) in IEC0
IEC0SET = 1 << 9;

// Clear bits 4..0 in IPC2 (old priority/subpriority)
IPC2CLR = 0b11111;

// Set priority=5 (0b101 in bits 2..0) and subpriority=2 (0b10 in bits
4..3)
IPC2SET = 0b10101;
```

b. Enable the Real-Time Clock and Calendar interrupt, set its flag status to 0, and set its vector's priority and subpriority to 6 and 1, respectively.

```
RTCCIF is on bit 15

// 1. Clear the RTCC interrupt flag (RTCCIF=0).
IFS1CLR = 1 << 15;

// 2. Enable the RTCC interrupt (RTCCIE=1).
IEC1SET = 1 << 15;

// 3. Clear the old priority/subpriority in IPC3's lower 5 bits (bits
4..0).
```

```
IPC3CLR = 0b11111;

// 4. Set priority=6 and subpriority=1 in those same 5 bits.
IPC3SET = 0b01110;
```

c. Enable the UART1 receiver interrupt, set its flag status to 0, and set its vector's priority and subpriority to 7 and 3, respectively.

```
// 1. Clear the UART1 RX interrupt flag (bit27)
IFS0CLR = 1 << 27;

// 2. Enable the UART1 RX interrupt (bit 27)
IEC0SET = 1 << 27;

// 3. Clear the old priority/subpriority in the lower 5 bits of IPC9
//    (assuming U4RX priority fields are bits 4..0 in IPC9)
IPC6CLR = 0b11111;

// 4. Set priority=7 and subpriority=3 => 0b11111
IPC6SET = 0b11111;
```

d. Enable the INT2 external input interrupt, set its flag status to 0, set its vector's priority and subpriority to 3 and 2, and configure it to trigger on a rising edge.

```
// 1. Clear the INT2 ext intrrupt flag (IFS0 bit 11)
IFS0CLR = 1 << 11;

// 2. enable the INT2 ext interrupt flag
IEC0SET = 1 << 11;

// 3. Clear the old priority/subpriority bits in IPC2<
IPC2CLR = 0b111111 << 24;

// 4. Priority = 3 (binary 011), Subpriority = 2 (binary 10)
IPC2SET = 0b01110 << 24;

// 5. Configure rising edge for INT2
INTCONSET = 1 << 2;
```

9. Edit Code Sample 6.3 so that each line correctly uses the "bits" forms of the SFRs. In other words, the left-hand sides of the statements should use a form similar to that used in step 5, except using INTCONbits, IPC0bits, and IEC0bits

See `hw6q9.c` for more details.C

16. Modify Code Sample 6.2 so the USER button is debounced. How can you change the ISR so the LEDs do not flash if the falling edge comes at the beginning of a very brief, spurious down pulse? Verify that your solution works.

See `hw6q16.c` for more details.

Debouncing strategy: I added a short delay in the beginning of the ISR and a countdown, during which any additional button press will be ignored and resets the interrupt status.

17. Using your solution for debouncing the USER button (Exercise 16), write a stopwatchprogram using an ISR based on INT2. Connect a wire from the USER button pin to the INT2 pin so you can use the USER button as your timing button. Using the PIC32 library, your program should send the following message to the user's screen: Press the USER button to start the timer. When the USER button has been pressed, it should send the following message: Press the USER button again to stop the timer. When the user presses the button again, it should send a message such as 12.505 seconds elapsed. The ISR should either (1) start the core timer at 0 counts or (2) read the current timer count, depending on whether the program is in the "waiting to begin timing" state or the "timing state." Use priority level 6 and the shadow register set. Verify that the timing is accurate. The stopwatch only has to be accurate for periods of less than the core timer's rollover time.

See `hw6q17.c` for more details. Also demo video in `q17demo`.